Simon Wroblewski

CMSI 402

February 20, 2019

**Homework #2**

**7.1**

The issue with these comments is that they are too obvious and don’t describe the code in any detail. Here is the code with additional comments.

// Use Euclid's algorithm to calculate the GCD.

// For more information visit en.wikipedia.org/wiki/Euclidean\_algorithm.

private long GCD( long a, long b )

{

a = Math.abs( a );

b = Math.abs( b );

for( ; ; )

{

long remainder = a % b;

// If remainder is 0, we're done. Return b.

If( remainder == 0 ) return b;

// Set a = b and b = remainder.

a = b;

b = remainder;

};

}

**7.2**

It seems the coder worked their way through the code in a top-down design, commenting things as they went along describing what the code did, resulting in the obvious comments. It’s also possible the code wasn’t commented until after it was written. Sometimes you get lazy and it’s not as hard to describe what the code does when its sitting right there.

**7.4**

The code should check and validate the inputs and results as well as have a Debug.Assert method to throw an error if anything is wrong.

**7.5**

In this case, the calling code should handle the errors. The code does not need error handling because the exceptions will be passed to the calling code as it is.

**7.7**

Walk to car.

Enter car.

Start car.

Pull out of parking space.

Descent from the parking structure.

Open property gate.

Go straight. Drive to the end of the street.

Turn right. Drive until you see a stop sign.

Turn right. Drive until you reach the traffic light.

Turn right. Drive until you arrive at the next traffic light.

Turn left. Drive until you arrive to the parking garage.

Turn right. Take a ticket to enter parking garage.

Put the ticket in a safe place.

Park the car in an open space.

Get out of the car.

Close and lock the car.

Enter supermarket.

Buy provisions.

**Assumptions:**

I have my keys.

Mirrors and seats can be adjusted to new driver.

Car has gas.

Driver knows how to drive.

There are available parking spots at the market.

The friend can remember what provisions to buy.

**8.1**

After some searching online I found some code that does this. The first method is Validate\_AreRelativelyPrime and is used to test the AreRelativelyPrime method.

// Return true if a and b are relatively prime.

// This is a test method that is used only to validate AreRelativelyPrime.

// See https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119209515.app1

private bool Validate\_AreRelativelyPrime(int a, int b) {

// Use positive values.

a = Math.Abs(a);

b = Math.Abs(b);

// If either value is 1, return true.

if ((a == 1) || (b == 1)) return true;

// If either value is 0, return false.

// (Only 1 and -1 are relatively prime to 0.)

if ((a == 0) || (b == 0)) return false;

// Loop from 2 to the smaller of a and b looking for factors.

int min = Math.Min(a, b);

for (int factor = 2; factor <= min; factor++)

{

if ((a % factor == 0) && (b % factor == 0)) return false;

}

return true;

}

Here is the Test for AreRelativelyPrime in pseudo code:

For 1,000 trials, pick random a and b and:

Assert AreRelativelyPrime(a, b) =

Validate\_AreRelativelyPrime(a, b)

For 1,000 trials, pick random a and:

Assert AreRelativelyPrime(a, a) =

Validate\_AreRelativelyPrime(a, a)

For 1,000 trials, pick random a and:

Assert AreRelativelyPrime(a, 1) relatively prime

Assert AreRelativelyPrime(a, -1) relatively prime

Assert AreRelativelyPrime(1, a) relatively prime

Assert AreRelativelyPrime(-1, a) relatively prime

For 1,000 trials, pick random a (not 1 or -1) and:

Assert AreRelativelyPrime(a, 0) relatively prime

Assert AreRelativelyPrime(0, a) relatively prime

For 1,000 trials, pick random a and:

Assert AreRelativelyPrime(a, -1,000,000) =

Validate\_AreRelativelyPrime(a, -1,000,000)

Assert AreRelativelyPrime(a, 1,000,000) =

Validate\_AreRelativelyPrime(a, 1,000,000)

Assert AreRelativelyPrime(-1,000,000, a) =

Validate\_AreRelativelyPrime(-1,000,000, a)

Assert AreRelativelyPrime(1,000,000, a) =

Validate\_AreRelativelyPrime(1,000,000, a)

Assert AreRelativelyPrime(-1,000,000, -1,000,000) =

Validate\_AreRelativelyPrime(-1,000,000, -1,000,000)

Assert AreRelativelyPrime(1,000,000, 1,000,000) =

Validate\_AreRelativelyPrime(1,000,000, 1,000,000)

Assert AreRelativelyPrime(-1,000,000, 1,000,000) =

Validate\_AreRelativelyPrime(-1,000,000, 1,000,000)

Assert AreRelativelyPrime(1,000,000, -1,000,000) =

Validate\_AreRelativelyPrime(1,000,000, -1,000,000)

**8.3**

Since the statement of Exercise 1 doesn’t say how the AreRelativelyPrime method works, this must be a black-box test. You could use and white-box or gray-box test if you knew how the AreRelativelyPrime method worked or if that information was stated.

**8.5**

There was some work to be done on the AreRelativelyPrime method. There were no limit on what values a or b could be so restricting these values would make for a good change. Since there were so many special cases, it requires more critical thinking which I think is a benefit of writing these tests.

**8.9**

Exhaustive tests fall under the black-box test category because they do not need to know what is going on inside the method that they are testing.

**8.11**

You can use each pair of testers to calculate three different Lincoln indexes.

* Alice/Carmen 5 \* 5 / 2 = 12.5
* Bob/Carmen 4 \* 5 / 2 = 10
* Bob/Alice 4 \* 5 / 1 = 20

After this you can either assume the worst and prepare for 20 bugs or average them and expect roughly 14 bugs.

**8.12**

If the two testers don't find any bugs in common, the equation for the Lincoln index divides by 0, giving a result of infinity which effectively results in you not knowing how many bugs there are. You can create a bound for the bugs by assuming 1 bug was found in common. For example, if the testers found 8 and 9 bugs, respectively, then the lower bound index would be (8 \* 9) / 1 = 72 bugs.